5. Езици за програмиране

Проф. д-р Емил Хаджиколев

1. Основни елементи в ЕП
2. Синтаксис и семантика
3. Видове грешки

Основни елементи в ЕП

В тази лекция ще разгледаме **основните елементи на ЕП** (подобни са на ЕЕ) и в частност на C++.

- **символи/знаци** от определена азбука (**от различни кодови таблици**);
- различни видове **думи**:
  - стандартни ***ключови думи*** – носещи различни видове смисъл;
  - ***специални символи*** – използвани с определени цели
    - оператори;
    - коментари;
    - разделители;
  - ***идентификатори;***
- **синтаксис**, **определящ правила за съставяне** на
  - ***имена (на променливи, константи, функции)***;
  - ***изрази***;
  - ***конструкции***;
  - програмни части – ***блокове***, ***подпрограми***;
  - и като цяло на кода на програмите.

За някои от елементите ще говорим подробно в следващи лекции (тук само ще спомена).

Кодови таблици и символи

- Символите, които записваме в текста на програмите са от определени кодови таблици. В различни кодови таблици един символ заема повече или по-малко памет (брой байтове, с които се представя).
- Кодовите таблици на символите(знак, character) описват множество двойки от вида „число-символ”. Те задават съответствие между символ и число, чрез което той се представя. Символите са букви от естествените езици, цифри, аритметични оператори, специални символи (интервал, tab, ins, del, край на ред...) и др.
- Един и същи символ може да има различни кодове в различни кодови таблици.
- При запис на конкретен символ (в текстов файл), в паметта се записва неговия числов код (във вид на нули и единици) от конкретна кодова таблица.
- За да бъде разчетена правилно дадена последователност от нули и единици, трябва да се знае каква кодовата таблица е използвана за кодирането ѝ. Чрез нея се определят групичките битове, сформиращи кода на един символ, а в зависимост от кода се изобразява съответния му символ.
- Текстообработващите програми скриват тези детайли от потребителите.

ASCII кодова таблица

- Първата компютърна кодова таблица е ASCII (American Standard Code for Information Interchange). Тя е 7-битова и дефинира записа на 128 стандартни символа, базирани на латинската азбука. Разширената 8 битовата версия (Extended ASCII) позволява записването на специфични езикови символи на избран естествен език. При това обаче, един и същи код може да има различни съответни символи за различните езици.
- Разработените след нея кодови таблици се съобразяват с основната 7-битова ASCII таблица и в тях първите 128 символа имат еднакви съответни цифрови кодове.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

- ASCII кодовата таблица, източник: http://www.cpptutor.com/ascii.htm
- Първите 32 символа са специални – някои от тях нямат изображения

UNICODE стандарт

- На базата на ASCII кодовата таблица са създадени **множество други кодови таблици** (описани в стандарти ISO-8859-n, Windows-n и др.), включващи освен основните 128 символа и символи от специфични езици.
- **ASCII символите, обикновено, запазват позициите си в различните кодови таблици**.
- Поради необходимостта от стандартизиране, в последствие е създаден **универсалния стандарт UNICODE**. Той е **многоезичен и описва символите на всички естествени езици, както и други специални символи**.
- **Символите в UNICODE заемат от 1 до 4 байта**.

Техники за кодиране на UNICODE символи

- **UTF-8, UTF-16, UTF-32** са различни техники за кодиране на UNICODE символите – на групи по 8, 16 или 32 бита (1, 2, 4 байта).
- И с трите кодировки се кодират всякакви UNICODE символ, но заеманата памет при различни символи може да е различна:
  - UTF-8 в зависимост от номера на символа заеманата памет е 1, 2, 3 или 4 байта;
  - UTF-16 – 2 или 4 байта;
  - UTF-32 – 4 байта.
- За ASCII символи е удачно да се ползва UTF-8, тъй като заемат 1 байт. Ако за тях ползваме например UTF-32, то ще се ползват 3 излишни байта за кодирането им.
- Ако в дадено приложение използваме символи с големи номера (от различни естествени езици) е по-добре да изберем UTF-16 или UTF-32.
- Ако паметта е ограничена – UTF-8. За бързодействие – UTF-16 или UTF-32.

Българските букви в UNICODE

1040 - А	1056 - Р	1072 - а	1088 - р
1041 - Б	1057 - С	1073 - б	1089 - с
1042 - В	1058 - Т	1074 - в	1090 - т
1043 - Г	1059 - У	1075 - г	1091 - у
1044 - Д	1060 - Ф	1076 - д	1092 - ф
1045 - Е	1061 - Х	1077 - е	1093 - х
1046 - Ж	1062 - Ц	1078 - ж	1094 - ц
1047 - З	1063 - Ч	1079 - з	1095 - ч
1048 - И	1064 - Ш	1080 - и	1096 - ш
1049 - Й	1065 - Щ	1081 - й	1097 - щ
1050 - К	1066 - Ъ	1082 - к	1098 - ъ
1051 - Л	1067 - Ы	1083 - л	1099 - ы
1052 - М	1068 - Ь	1084 - м	1100 - ь
1053 - Н	1069 - Э	1085 - н	1101 - э
1054 - О	1070 - Ю	1086 - о	1102 - ю
1055 - П	1071 - Я	1087 - п	1103 - я

Шрифтове

- Шрифтовете определят графични представяния на символите от кодовите таблици. Един символ може да има различни графични представяния за различните шрифтове.
- Шрифтовете имат характеристики, чрез които може да се променя изображението им: размер на буквите, тегло (normal, bold и др.), наклон (normal, italic и др.) и др. Не е задължително един шрифт да има графично представяне за всички символи, които могат да се запишат чрез използваната кодова таблица.
- Някои по-известни шрифтове са Times New Roman, Verdana, Arial, Courier...

ЕП и кодови таблици

- При различни ЕП се поддържат като основни различни кодови таблици.
- В първите версии на езика C++ се поддържа само ASCII кодовата таблица.
- Новите версии на C++ и създадените по-късно езици (Java, C# и др.) се поддържа UNICODE като базова кодова таблица.
- Възможно е да има проблеми с българския език в C++.
  - При някои компилатори и среди (напр., https://www.onlinegdb.com/online_c++_compiler) няма необходимост от специфични настройки.
  - Във Visual Studio – се ползват някои функции за българския език.

Български език за VisualStudio – Вариант 1 (Само изход)

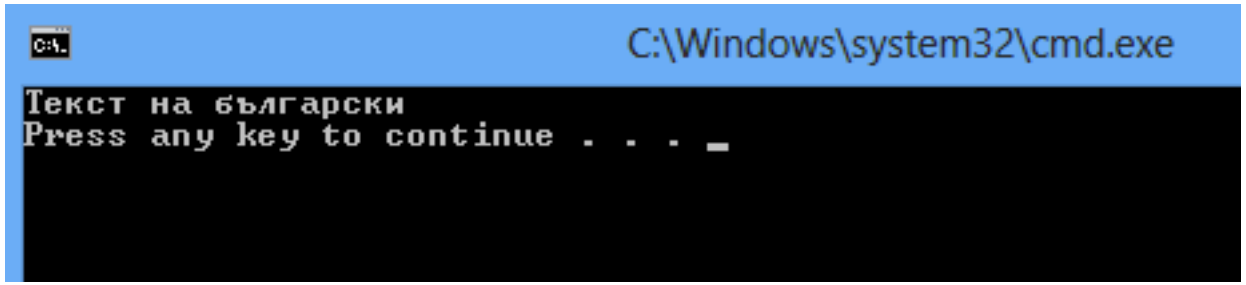
```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    setlocale(LC_ALL, "Bulgarian");
    // стандартна функция за задаване на локализация - на български език в случая

    cout << "Текст на български\n";

    return 0;
}
```



Български език за VisualStudio – Вариант 2 (вход и изход)

```cpp
#include <iostream>
#include <windows.h>
#include <string>

int main() {
    SetConsoleOutputCP(1251);
    SetConsoleCP(1251);

    std::string name;
    std::cout << "Въведи име: ";
    std::getline(std::cin, name);

    std::cout << "Здравей, " << name << "!\n";
    return 0;
}
```

Или по-универсално, но с нужда от допълнителни настойки

```cpp
SetConsoleOutputCP(CP_UTF8);
SetConsoleCP(CP_UTF8);
```

Думи в ЕП

- стандартни ***ключови думи*** – носещи конкретен семантичен смисъл;
- ***специални символи*** – използвани с определени цели
  - оператори;
  - коментари;
  - разделители;
- ***идентификатори***.

Ключови думи

- **Ключовите думи определят смисъла на специфични части от кода**, записан обикновено след тях.
- Например, ключовата дума „int“ указва тип, след който трябва да се зададе (не винаги е задължително) име на променлива: „int age“ означава, че сме декларирали променлива с име age от тип за цяло число int.

Ключови думи в C++ (https://cppreference.com/w/cpp/keywords.html)

alignas (C++11)
alignof (C++11)
and
and_eq
asm
atomic_cancel (TM TS)
atomic_commit (TM TS)
atomic_noexcept (TM TS)
auto (1) (3) (4) (5)
bitand
bitor
bool
break
case
catch
char
char8_t (C++20)
char16_t (C++11)
char32_t (C++11)
class (1)
compl
concept (C++20)
const
consteval (C++20) (5)
constexpr (C++11) (3)
constinit (C++20)
const_cast
continue
contract_assert (C++26)
co_await (C++20)
co_return (C++20)
co_yield (C++20)

decltype (C++11) (2)
default (1)
delete (1)
do
double
dynamic_cast
else
enum (1)
explicit
export (1) (4)
extern (1)
false
float
for (1)
friend
goto
if (3) (5)
inline (1) (3)
int (1)
long
mutable (1)
namespace
new
noexcept (C++11)
not
not_eq
nullptr (C++11)
operator (1)
or
or_eq
private (4)
protected
public

reflexpr (reflection TS)
register (3)
reinterpret_cast
requires (C++20)
return
short
signed
sizeof (1)
static
static_assert (C++11)
static_cast
struct (1)
switch
synchronized (TM TS)
template
this (5)
thread_local (C++11)
throw (3) (4)
true
try
typedef
typeid
typename (3) (4)
union
unsigned
using (1) (4)
virtual
void
volatile
wchar_t
while
xor
xor_eq

Основни групи ключови думи

модификатори за достъп	private, protected, public
управляващи конструкции	break, case, continue, default, do, else, for, if, return, switch, while
примитивни типове	bool, char, wchar_t, char16_t, char32_t, double, float, int, long, short, signed, unsigned, void
декларации на съставни типове и области	class, struct, union, enum, typedef, namespace
прехващане на грешки	try, catch, throw
други модификатори	const, virtual, volatile, inline, mutable, extern
други	this, true, false, new, delete

- Ще разгледаме част от ключовите думи в следващи лекции
- За част от ключовите думи има съответни оператор: and-&&, or-|| и др.

Оператори и операции – терминология

- **Операторите извършват специфични действия върху обекти** (наречени операнди) от подходящи типове.
- **Действията, които се извършват от операторите се наричат операции**.
- т.е. **Операторът извършва операция върху операнди**.
- Например, символът ‚+‘ в математиката и езиците за програмиране е оператор за извършване на операцията сумиране, която се прилага върху два операнда-числа.
- **Операндите понякога се наричат и аргументи**.
- В различни литературни източници (на български език), операторите се наричат операции (а думата оператор се използва за команда или конструкция).

Основни оператори в C++ (https://cppreference.com/w/cpp/language/expressions.html#Operators)

Основни оператори в C++						
За присвояване	Increment и decrement (увеличаване/намаляване с единица)	Аритметични (и побитови)	Логически	За сравнение	Достъп до поделемент (member)	Други

`a = b`
`a += b`
`a -= b`
`a *= b`
`a /= b`
`a %= b`
`a &= b`
`a |= b`
`a ^= b`
`a <<= b`
`a >>= b`
	`++a`
	`--a`
	`a++`
	`a--`
		`+a`
		`-a`
		`a + b`
		`a - b`
		`a * b`
		`a / b`
		`a % b`
		`~a`
		`a & b`
		`a | b`
		`a ^ b`
		`a << b`
		`a >> b`
			`!a`
			`a && b`
			`a || b`
				`a == b`
				`a != b`
				`a < b`
				`a > b`
				`a <= b`
				`a >= b`
					`a[b]`
					`*a`
					`&a`
					`a->b`
					`a.b`
					`a->*b`
					`a.*b`
						`a(...)`
						`a, b`
						`? :`

Специални оператори в C++ (http://en.cppreference.com/w/cpp/language/expressions#Operator)

В лекциите ще разгледаме само някои от специалните оператори.

Специални оператори в C++
static_cast converts one type to another related type
dynamic_cast converts within inheritance hierarchies
const_cast adds or removes cv qualifiers
reinterpret_cast converts type to unrelated type
C-style cast converts one type to another by a mix of static_cast, const_cast, and reinterpret_cast
new allocates memory
delete deallocates memory
sizeof queries the size of a type
sizeof... queries the size of a parameter pack (since C++11)
typeid queries the type information of a type
noexcept checks if an expression can throw an exception (since C++11)
alignof queries alignment requirements of a type (since C++11)

Оператори и изрази

- Има **оператори** с един, два или три аргумента (операнда). Те се наричат съответно **унарни (unary), бинарни (binary), тройни (ternary)**.
- **Израз е комбинация от оператори и операнди**. Резултатът от изчислението на израз е стойност от някакъв тип – за число, низ или друг. Операндите могат да бъдат както конкретни литерали, така и променливи, константи или функции, получаващи стойност по време на изпълнение на програмата.
- **Начинът на изчисляване на един израз зависи от приоритета и асоциативността на участващите оператори.**

Приоритет на операторите

- **Приоритетът определя реда на прилагане на операторите.** Операторите с по-висок приоритет се изчисляват преди тези с по-нисък. Например, при аритметичните оператори, с по-висок приоритет са *, /, %, а с по-нисък + и -. Приоритетът може да се промени като се използват скоби.
- В следната таблица може да се види резултатът от изчислението на елементарен израз, с и без използване на скоби.

№	Израз	Резултат
1.	1+2*3	7
2.	(1+2)*3	9
3.	1+(2*3)	7

- Изрази 1 и 3 са еквивалентни. В зависимост от конкретната задача, която трябва да се реши, всеки един от изразите може да е правилен.
- Ако не сме сигурни в приоритетите на операторите е добре да използваме скоби, за да не възникват семантични грешки.

Асоциативност на операторите

- **Асоциативността определя реда на присъединяване на операндите към оператора.**
- **При повечето оператори, първо се взима/изчислява стойността на левия операнд, а след това на десния**. Те се наричат **ляво-асоциативни**.
- **При унарните оператори, обаче, има само един десен операнд и първо се изчислява неговата стойност**. Такава е логиката **и при оператора за присвояване: първо се изчислява изразът в дясно, а след това той се присвоява на променливата в ляво**. Унарните оператори и операторите за присвояване са **дясно-асоциативни**.

По-подробно за приоритет и асоциативност на операторите ще говорим в следваща лекция.

Коментари

- Коментарите описват алгоритъма, който се реализира в програмата и отделните подпрограми, както и смисъла на дадена величина отделен ред или група от редове на кода.
- Коментарите се използват единствено от хората, които пишат или разглеждат изходния (source) код на програмата. Целта е да се подпомагат бъдещите промени на кода.
- В някои ЕП (Java, C#) от коментарите и кода може да се генерира автоматично (със стандартни инструменти) документация (във вид на уеб страници), която улеснява четенето и разбирането на библиотеките и възможностите, които те предлагат. За C++ няма стандартизирани инструменти (може да се ползва Doxygen).
- Коментарите не участват в изпълнимия код на програмата.

Коментари в C++

```
/* Това е коментар,
 записан на няколко реда.
*/

// това е коментар на един ред
```

- Коментарите на един ред започват с // и завършват в края на реда (със символ за край на ред – enter – който не се вижда)
- Коментарите на няколко реда започват със символите /* и завършват със символите */

Бели полета (вид разделители)

- Белите полета са символи, които служат за форматиране на кода и разделяне на думите една от друга. Те са два основни вида:
- **Край на ред** (Line Terminators) – използват се ASCII символите LF (с код 10 – newline), CR (с код 13 – return) или комбинация от двата. В зависимост от текстовия редактор и операционната система при натискане на клавиша „Enter” се записва някой от вариантите.
- **Бяло поле** (White Space) – интервал (код 32) и таб (код 9).

Разделители в ЕП (и C++)

Използват се за разделяне на специални части от кода:

- ( ) – заграждат формалните и фактическите параметри при деклариране и извикване на функции; участват в математически изрази; използват се при преобразуване по тип и при оператори за контрол на изпълнението на кода;
- { } – при работа с масиви и дефиниране на блокове;
- [ ] – при работа с масиви;
- <> – при работа с шаблонни типове (generics, templates)
- ; – край на команда;
- , – разделя променливи при декларация;
- " – текст заграден в кавички е низ;
- ' - апостроф, за ограждане на символи.

Идентификатори

- Идентификаторите задават уникални имена за елементи на програмата – функции, променливи, константи и др.
- След дефинирането им елементите могат многократно да бъдат използвани чрез името си.
- Идентификаторите в ЕП и (C++):
  - може да съдържат букви (може и на български, но зависи от версията на C++ или използвания ЕП) , цифри, _ (подчертаващо тире);
  - може да започват с буква или _;
  - не може да започват с цифра;
  - не може да съдържат специални символи (бели полета, разделители и основни оператори). В C++ само, оператори могат да бъдат предефинирани. Тогава има изключение от това правило – създава се функция със специално име operatorX, където X е съществуващ оператор...
  - не може да са ключови думи (защото те се интерпретират по специални правила), но може да съдържат като част от името ключова дума.
- В C++ (и други ЕП) се прави разлика между малки и големи букви – напр., идентификатор „age“ е различен от „Age“ или „aGe“.

Стилове/конвенции за именуване

- Добре е имената да са смислени – може да са съставени и от няколко думи;
- Константите са с големи букви;
- Основни стилове при идентификатори от няколко думи:
  - lowerCamelCase – имена на променливи и функции – първата дума започва с малка буква, следващите думи – с голяма, всички останали букви са малки.
  - UpperCamelCase, PascalCase – съставни типове (класове, структури и др.) – всички думи започват с главна буква, а останалите букви са малки.
  - snake_case – думите се разделят с „_“
    - ползва се за именуване най-вече за константи, които се задават с главни букви;
    - може и за променливи, типове, функции (но не е много използван подход в тези случаи);
  - Унгарска нотация – пред името се добавя и типа – intX, strName;
  - изключения от горните стилове при абревиатури – XMLNode, HTTPRequest…

Примери за идентификатори

Добър стил, правилни	Лош стил, правилни	Неправилни

int age = 2;
string first_name = "Иван";
string firstName = "Иван";
	int Age = 2;
	string parvoime = "Иван";
	string parvo_ime = "Иван";
	string firstname = "Иван";
	string t= "Иван"; // това е променлива за име
		string 1name = "Иван";
		string first-name = "Иван";
		string first name = "Иван";

Защо...?

Литерали

Литералите са част от думите в ЕП. Те представят числови, булеви и други стойности от различни типове в кода на програмата. Ще ги разгледаме по-подробно в следващи лекции. Основните видове са:

- Числа – описани по стандартни начини чрез цифри и други символи – **0, 1.1, 2.2E4(=22000), 123456789012345678 9LL** (число от тип long long int);
- Низ – текст, заграден в кавички – **"Текст на български\n"**;
- Символ – символ, заграден в апострофи - **'s'**;
- Литерали за булеви стойности – **true, false**;
- **NULL** – литерал (макрос), за задаване на стойност на динамични обекти. Означава „неизвестна стойност“ или „нулев адрес“. Съответства на различни нулеви стойности – 0, nulptr, false.

Величини

- Величините представят данни.
- **Величините служат за съхранение на стойности от определени типове.**
- Имат **три основни характеристики**:
  - ***тип;***
  - ***име/идентификатор (за някои видове величини);***
  - ***стойност.***
- **Величините са два вида:**
  - ***Променливи;***
  - ***Константи;***
- В зависимост от това, къде се използват величините, могат да имат специфични наименования – поле на клас/структура, променлива, константа, формален параметър на функция/метод, фактически параметър на функция/метод.

Променливи. Пример

Променливите могат да променят стойността си по време на изпълнение на програмата. Имат име, чрез което става обръщение към тях. Биват два вида:

- **Вътрешни** – контролират се от програмата;
- **Външни** – стойността им се определя от средата, в която се изпълнява програмата. Напр. има променлива за системното време.

```
int year = 2000;
// int е тип; year е променлива от тип int; стойността на year е 2000;
// 2000 е литерал;
year = year + 1;
// изчислява се израза от дясно на оператора „=“, защото той е
// дясно-асоциативен. Получава се 2001 и стойността му и се
// присвоява като нова стойност на променливата year;
// 1 е литерал;
```

Константи. Пример

Константи – стойността им се задава еднократно. Те са два вида:

- **Именувани константи** – подобно на променливите – имат име, но не могат да променят стойността си;
- **Литерали** – записана директно в кода на програмата стойност, която няма съответно име.

```
const double PI = 3.14;
// създаване на константа - const – PI, от тип за
// реални числа double, със стойност 3.14;
// 3.14 е литерал;
PI = 3.14159265359;
// грешка - не може да се присвоява нова стойност на константа;
// 3.14159265359 е литерал;
```

Типове

- **Типът определя** дефиниционна област на величините, т.е. **множество от възможни стойности, които могат да се присвояват на величините**.
- Всеки тип се характеризира с
  - име
  - размер на паметта (брой битове), необходими за запис на величина от съответния тип.
- В ЕП съществуват **два вида типове**
  - **примитивни** (прости, елементарни)
  - **съставни** (сложни)**.**

Примитивни типове

- **Примитивните типове са четири вида**:
  - за цели числа – ***int;***
  - за реални числа с плаваща запетая – ***float, double;***
  - за булеви стойности – ***bool;***
  - за символи – ***char*** (и други разновидности)***.***
- **При типа int** може да се укажат **модификатори за знак:**
  - ***signed*** – със знак
  - ***unsigned*** – без знак
- За int може да се указва допълнително и размера (байтовете) с помощта на ключовите думи short и long

Съставни типове

- **Съставните типове обикновено комбинират няколко величини от различни типове и/или методи**.
- **Съставните типове в C++ са**:
  - ***масив;***
  - ***структура;***
  - ***обединение;***
  - ***изброим тип;***
  - ***клас.***
- В други ЕП има различни базови съставни типове – допълнителни (като интерфейси) и/или липсващи (структура, обединение).

Тип string

- **Специален съставен тип в C++ е класът string за работа с низове**. Той донякъде прилича на примитивен тип, тъй като за него са създадени специални оператори за присвояване на стойност (=), конкатениране (слепване) на низове (+), сравнение на два низа.
- За да използваме типът string, в началото на нашата програма трябва да включим библиотеката string:

```
#include <string>
```

Синтаксис и семантика

- Синтаксисът на ЕП (и на ЕЕ) определя как може да се комбинират различните елементи на езика.
- **Синтаксисът задава правила за създаване на коректни езикови конструкции.**
- **Семантиката определя какъв е смисъла на правилните езикови конструкции** т.е. как те се разбират от хората и от машините.
- Възможно е една програма да е написана синтактично вярно, но смисълът ѝ да е грешен.

Видове грешки в програмите

- **Синтактични** – грешки, при които не са спазени правилата за писане на код на програмите – напр. декларацията на променлива и други езикови конструкции завършват задължително със символа ‘;’. При липсата му програмата се счита за грешна и не може да бъде компилирана/интерпретирана.
- **Семантични** (логически, смислови) – грешки, които не могат да бъдат уловени от транслатора. При такива грешки програмата извежда грешни резултати при изпълнение (в run-time):
  - винаги;
  - понякога - случайно може да изведе и верен резултат.
  - Напр., семантична грешка е “произведението на x и y” да запишем като “x/y” (за x=1 и y=1 ще се получи верен резултат).

Декларация, дефиниция, инициализация на променлива (1)

- **Променлива** в повечето ЕП и C++ **се декларира със следния синтаксис**:

  <тип> <идентификатор_на_променлива>;

  - В синтаксиса, с ъглови скоби (<>) се описват задължителните елементи.
  - За **тип** се задава идентификатор на **съществуващ примитивен или съставен тип**.
  - За **идентификатор_на_променлива** задаваме сами, като той трябва да **отговаря на условията за съставяне на идентификатори**.
  - **Конструкцията** (синтактичната конструкция за декларация на променлива) **приключва с точка и запетая** (;).

Декларация, дефиниция, инициализация на променлива (2)

- Задаването на стойност на променлива се извършва с оператора за присвояване „=” или в скоби след името.
- **Ако едновременно с декларацията на променлива ѝ се задава стойност, декларацията се нарича дефиниция;**
- **Първоначалното задаване на стойност на променлива се нарича инициализация; т.е. дефиницията включва инициализация;**
- В различни ЕП, при декларация променливата автоматично се инициализира (например 0 за числовите типове, false за булевите).
- Използването на неинициализирана променлива, обикновено, води до грешка при компилиране.
- **В C++ променливите от примитивен тип не се инициализират автоматично при декларация** – но различни компилатори може да правят това.
- В C++ при съставен тип се прави автоматична инициализация на поделементи (което ще разгледаме малко по-подробно в следваща лекция).

Декларация, дефиниция, инициализация на променлива в C++. Пример (1)

```
setlocale(LC_ALL, "Bulgarian");
// стандартна функция за задаване на локализация на български език

double d;                          // декларация
//cout << "d = " << d << "\n";     // грешка - използва се неинициализирана променлива
d = 4.5;                           // първоначално задаване на стойност - инициализация
cout << "d = " << d << "\n";       // стойността на d е 4.5
d = 5.5;                           // задаване на стойност - не е инициализация
cout << "d = " << d << "\n";       // стойността на d е 5.5

float f = 3.5;                     // дефиниция - включва инициализация
cout << "f = " << f << "\n";       // стойността на f е 3.5
```

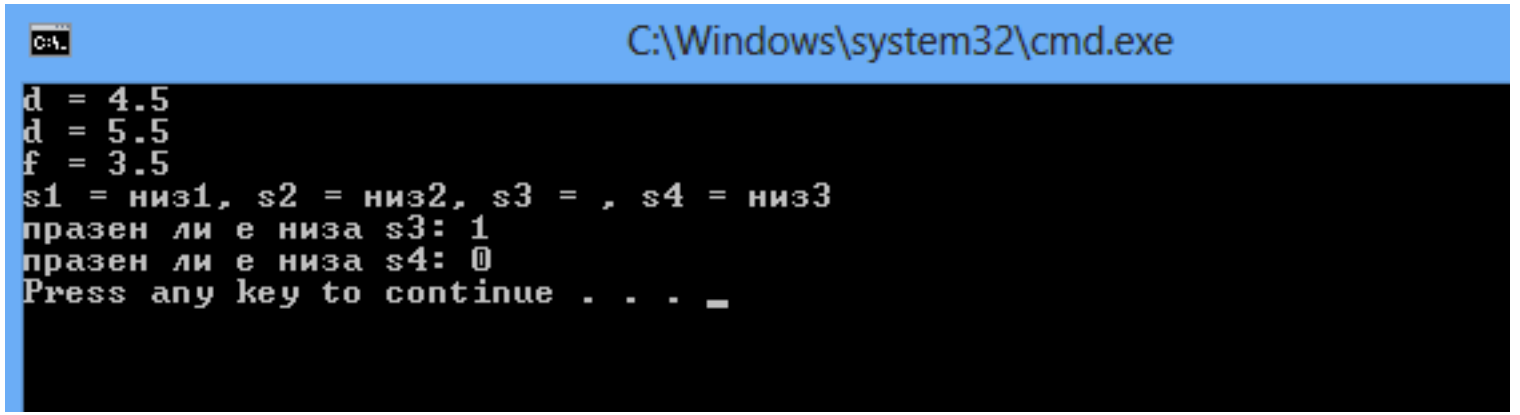
Декларация, дефиниция, инициализация на променлива в C++. Пример (2)

```
string s1 = "низ1";// дефиниция - неявно използване на конструктор на класа string
string s2 = string("низ2"); // дефиниция - явно използване на конструктор на класа
string s3 = "";             // празен низ
string s4;                  // празен низ
s4 = "низ3";                // задаване на нова стойност

cout << "s1 = " << s1 << ", s2 = " << s2 << ", s3 = " << s3 << ", s4 = " << s4 << "\n";
// s1 = низ1, s2 = низ2, s3 = , s4 = низ3

cout << "празен ли е низа s3: " << (s3 == "") << "\n";  // 1 - съответства на true
cout << "празен ли е низа s4: " << (s4 == "") << "\n";  // 0  - съответства на false
```

Декларация, дефиниция, инициализация на променлива в C++. Пример (3)



Едновременна декларация на няколко променливи от един и същи тип

- Синтаксис:

  <тип> <списък_с_идентификатори>;

където списък_с_идентификатори е

  <идентификатор_1>, <идентификатор_2>…, <идентификатор_n>

- Пример

```
double d1, d2;
```

- По време на декларацията може да се задават стойности (с оператор „=“ или в скоби след името):

```
double d1=3.1, d2(4.1);
```

Дефиниция на именувани константи

- Именуваните константи се дефинират подобно на променливите но с **модификатор const** пред типа:

  const <тип> <идентификатор_на_константа> = <стойност>;

- Задължително, стойност на константата се задава при декларацията на променлива.
- Пример:

```
const double PI = 3.14159265359;
// дефиниция на константа
// след това (в кода) може само да се използва
```

Стил за именуване на константи

- Прието е константите да се записват с главни букви (за да се различават от променливите);
- Ако идентификаторите се състоят от повече думи, те се отделят с подчертаващо тире (_).
- Примери:

Добър стил	Лош стил
`const double PI = 3.14159265359;`	`const double pi = 3.14159265359;`
`const string COLOR_BLACK = "black";`	`const string color_black = "black";`
`const string COLOR_WHITE = "white";`	`const string colorWhite = "white";`

Използване на променливи, литерали, именувани константи

Ако на няколко места в кода се налага да използваме една и съща константа, кой от следните два кода е по добър и защо?

Код 1:

```
const double PI = 3.14159265359; // константа PI
double r = 3;                    // радиус

cout << "Лице на кръг с радиус " << r << " е " << PI * r * r << "\n";
cout << "Обиколка на кръг с радиус " << r << " е " << 2 * PI * r << "\n";
```

Код 2:

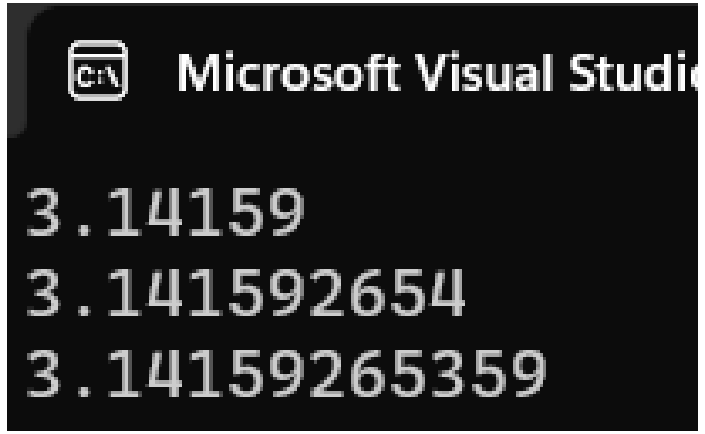
```
cout << "Лице на кръг с радиус 3 е " << 3.14159265359 * 3 * 3 << "\n";
cout << "Обиколка на кръг с радиус 3 е " << 2 * 3.14159265359 * 3 << "\n";
```

Прецизност на реалните числа

- В C++ реалните числа се показват с определена точност/прецизност и не винаги се виждат всички цифри. Това може да се промени с манипулация на изхода (библиотека iomanip), което няма да разясняваме в детайли:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    const double PI = 3.14159265359;
    cout << PI << endl;
    cout << setprecision(10) << PI << endl;
    cout << setprecision(15) << PI << endl;
    return 0;
}
```



Управляващи конструкции (конструкции за контрол на изпълнението)

- В кода на програмите се описват последователно (линейно) различни команди. Изпълнението им обаче, в общия случай не е последователно.
- За да променим последователността на изпълнение на кода използваме управляващи конструкции:
  - Чрез **условни конструкции if, if-else**, и **конструкция за избор от варианти switch-case** се изпълнява код само при определени условия;
  - Чрез **конструкция за цикъл while, do-while и for** даден код се изпълнява многократно.
- Чрез управляващите конструкции в програмите се представят разклонени и циклични алгоритми.

Ще ги разглеждаме в следващи лекция.

Синтаксис на декларация на подпрограма (1)

- Декларацията на функция в C++ има следния (основен) синтаксис

  <тип_на_връщана_стойност>
  <идентификатор_на_функция>([<списък_с_формални_параметри>]) ;

където в квадратни скоби са описани незадължителните елементи

- **тип_на_връщана_стойност** може да е всеки съществуващ тип.
- **идентификатор_на_функция** спазва правилата за именуване.
- скобите – () – са задължителни, без значение дали има или не формални параметри.

Синтаксис на декларация на подпрограма (2)

- синтаксисът на **списък_с_формални_параметри** е:

<тип> [const][volatile]<идентификатор_на_променлива_1>,… <тип> [const][volatile]<идентификатор_на_променлива_n>

- Формалните параметри
  - се декларират като променливи и са вид променливи;
  - валидни са (виждат се) само в тялото (блока) на функцията;
  - получават стойност при изпълнение (извикване) на функцията;
- Ако за променлива (в частност формален параметър) е зададена ключовата дума const, то в указания блок е забранено кодът да променя стойността ѝ. Декларацията на формален параметър като const предпазва от случайни грешки при програмирането: възможно е един човек да проектира дизайна на системата (функциите), а друг да програмира.
- Ако за променлива се укаже, че е volatile (променяща се) то тя може да променя стойността си от процес извън текущия код.
- Възможно е константа (const) да е в същото време и volatile: const означава, че кодът не може да променя стойността в текущия блок, но код извън блока (напр., при паралелни и/или хардуерни процеси) може да променя стойността ѝ.

Дефиниция на функция

- **Дефиницията на функция**
  - **включва декларацията** и
  - описанието на валиден код между отваряща и затваряща фигурни скоби – **{тяло}**.
  - **свързва името на функция с тялото й**;
- В C++ се допуска, само да се декларира функция (напр., в header-файлове), а в последствие тя да бъде дефинирана. В такъв случай в декларацията не е необходимо да се описват имената на формалните параметри.

Извикване на функция

- Синтаксисът за извикване на функция е:

  <име_на_функция>(<списък_с_фактически_параметри>)

- Списъкът с фактически параметри съдържа конкретни параметри, чрез които се задава стойност на съответните формални параметри.
- **Фактическите параметри може да са променливи, константи, изрази, други функции, които трябва да имат тип съвместим с типа на съответните формални параметри.**

Параметри по подразбиране

- Възможно е да задаваме стойности по подразбиране за формалните параметри на функции, като след декларацията запишем =<стойност>
- Задължително, параметрите със стойности по подразбиране се слагат в края на списъка. **В следващата декларация втория и третия параметър имат стойности по подразбиране**:

  ```
  void testDefaultValues(int i, int i1=1, int i2=2)
  ```

  Не може да запишем, например следното

  ~~`void testDefaultValues(int i1=1, int i)`~~
- Чрез параметрите по подразбиране една функция може да бъде извиквана с различен брой фактически параметри.

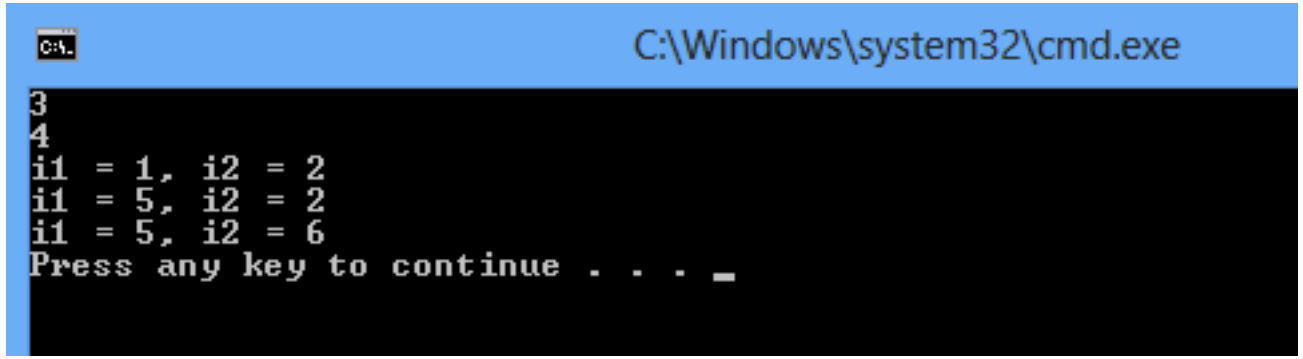
Пример за работа с функции (1)

```
// декларация и дефиниция на функция с име testConst и формален параметър,
// който не може да се променя в тялото
void testConst(int const i) {
     // i = 4;  // грешка - не може да се променя
     cout << i << "\n"; // 3
}
// декларация и дефиниция на функция с име testNormal и формален параметър,
// който  може да се променя в тялото
void testNormal(int i) {
     i = 4;
     cout << i << "\n"; // 4
}
// дефиниция на функция с параметри по подразбиране
void testDefaultValues(int i1=1, int i2=2) {
         cout << "i1 = " << i1 << ", i2 = " << i2 << "\n";
}
```

Пример за работа с функции (2)

```
int main(){
    testConst(3);              // извикване с фактически параметър 3 - извежда 3
    testNormal(3);             // извикване с фактически параметър 4 - извежда 4
    testDefaultValues();       // използват се стойностите по подразбиране
    testDefaultValues(5);      // за първият параметър задаваме 5,
                               // вторият получава стойността по подразбиране
    testDefaultValues(5, 6); // не се използват стойностите по подразбиране

    return 0;
}
```



Предефиниране (overload) на функциите в C++

- **Функции с едно и също име се наричат предефинирани.**

```
void testNormal(int i)
void testNormal()
```

- В C++ може да имаме функции с еднакви имена, но с различен брой параметри.
- Съответно, **не може да имаме функции с едно и също име и еднакъв брой параметри**, дори параметрите да са от различни типове

```
void testNormal(int i)
void testNormal(double d)
void testDefaultValues(int i1=1, int i2=2)
void testDefaultValues() – защо не може?
```

- В различни ЕП има различни сигнатури, идентифициращи по уникален начин функция/метод и ограниченията за предефиниране са различни. В Java, например, може да има методи с еднакви имена и еднакъв брой параметри, които обаче са от различен тип (при отделните методи).

Програма

- Програмата в C++
  - е функция, от която започва изпълнението на кода.
  - има специална декларация int main().
- Изпълнение на програма
  - конструкциите в програмата се изпълняват последователно;
  - ако се срещне управляваща конструкция, се изпълняват указаните от нея действия;
  - ако се срещне извикване на подпрограма, започват да се изпълняват указаните в тялото ѝ команди, а след приключване на изпълнението управлението се предава на извикващата функция.